Failing Securely

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-12-05

When a system fails, it should do so securely. This typically involves several things: secure defaults (default is to deny access); on failure undo changes and restore to a secure state; always check return values for failure; and in conditional code/filters make sure that there is a default case that does the right thing. The confidentiality and integrity of a system should remain even though availability has been lost. Attackers must not be permitted to gain access rights to privileged objects during a failure that are normally inaccessible. Upon failing, a system that reveals sensitive information about the failure to potential attackers could supply additional knowledge for creating an attack. Determine what may occur when a system fails and be sure it does not threaten the system.

Detailed Description Excerpts

According to [Saltzer 75] in "Basic Principles of Information Protection" on page 8:

> Fail-safe defaults: Base access decisions on permission rather than exclusion. This principle, suggested by E. Glaser in 1965,[8] means that the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted. The alternative, in which mechanisms attempt to identify conditions under which access should be refused, presents the wrong psychological base for secure system design. A conservative design must be based on arguments why objects should be accessible, rather than why they should not. In a large system some objects will be inadequately considered, so a default of lack of permission is safer. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situation, since it will be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use. This principle applies both to the outward appearance of the protection mechanism and to its underlying implementation.

According to Viega and McGraw [Viega 02] in Chapter 5, "Guiding Principles for Software Security," in "Principle 3: Fail Securely" on pages 97-100:[9]

> Any sufficiently complex system will have failure modes. Failure is unavoidable and should be planned for. What is avoidable are security problems related to failure. The problem is that when many systems fail in any way, they exhibit insecure behavior. In such systems, attackers only need to cause the right kind of failure or wait for the right kind of failure to happen. Then they can go to town.

3. daisy:35 (Barnum, Sean)

4. daisy:345 (Gegick, Michael)

8. In this material we have attempted to identify original sources whenever possible. Many of the seminal ideas, however, were widely spread by word of mouth or internal memorandum rather than by journal publication, and historical accuracy is sometimes difficult to obtain. In addition, some ideas related to protection were originally conceived in other contexts. In such cases, we have attempted to credit the person who first noticed their applicability to protection in computer systems, rather than the original inventor.

9. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

The best real world example we know is one that bridges the real world and the electronic world--credit card authentication. Big credit card companies such as Visa and MasterCard spend lots of money on authentication technologies to prevent credit card fraud. Most notably, whenever you go into a store and make a purchase, the vendor swipes your card through a device that calls up the credit card company. The credit card company checks to see if the card is known to be stolen. More amazingly, the credit card company analyzes the requested purchase in context of your recent purchases and compares the patterns to the overall spending trends. If their engine senses anything suspicious, the transaction is denied. (Sometimes the trend analysis is performed off line and the owner of a suspect card gets a call later.)

This scheme appears to be remarkably impressive from a security point of view, that is until you note what happens when something goes wrong. What happens if the line to the credit card company is down? Is the vendor required to say, "I'm sorry, our phone line is down"? No. The credit card company still gives out manual machines that take an imprint of your card, which the vendor can send to the credit card company for reimbursement later. An attacker need only cut the phone line before ducking into a 7-11.

There used to be some security in the manual system, but it's largely gone now. Before computer networks, a customer was supposed to be asked for identification to make sure the card matched a license or some other id. Now, people rarely get asked for identification when making purchases; we rely on the computer instead. The credit card company can live with authenticating a purchase or two before a card is reported stolen; it's an acceptable risk. Another precaution was that if your number appeared on a periodically updated paper list of bad cards in the area, the card would be confiscated. Also, the vendor would check your signature. These techniques aren't really necessary anymore, as long as the electronic system is working. If it somehow breaks down, then, at a bare minimum, those techniques need to come back into play. In practice, they tend not to, though. Failure is fortunately so uncommon in credit card systems that there is no justification for asking vendors to remember complex procedures when it does happen. That means when the system fails, the behavior of the system is less secure than typical behavior. How difficult is it to make the system fail?

Why do credit card companies use such a brain dead fallback scheme? The answer is that the credit card companies are good at risk management. They can eat a fairly large amount of fraud, as long as they keep making money hand over fist. They also know that the cost of deterring this kind of fraud would not be justified, since the amount of fraud is relatively low (there are a lot of factors considered in this decision, including business costs and public relations issues).

Plenty of other examples are to be found in the digital world. Often, the insecure failure problem occurs because of a desire to support legacy versions of software that were not secure. For example, let's say that the original version of your software was broken, and did not use encryption to protect sensitive data at all. Now you want to fix the problem, but you have a large user base. In addition, you have deployed many servers that probably won't be upgraded for a long time. The newer, smarter clients and servers need to interoperate with older clients that don't use the new protocols. You'd like to force old users to upgrade, but you didn't plan for that. Legacy users aren't expected to be such a big part of the user base that it will really matter, anyway. What do you do? Have clients and servers examine the first message they get from the other, and figure out what's going on from there. If we are talking to an old piece of software, then we don't perform encryption.

Unfortunately, a wily hacker can force two new clients to think each of the others is an old client by tampering with data as it traverses the network. (A form of man-in-the-middle-attack.) Worse yet, there's no way to get rid of the problem while still supporting full (two-way) backwards compatibility.

A good solution to this problem is to design in a forced upgrade path from the very beginning. One way is to make sure that the client can detect that the server is no longer supporting it. If the client can securely retrieve patches, it will be forced to do so. Otherwise, it tells the user that a new copy must be obtained manually. Unfortunately, it's important to have this sort of solution in place from the very beginning. That is, unless you don't mind alienating early adopters.

> We discussed a problem similar to the one in Chapter 3 [in *Building Secure Software*], which exists in most implementations of Java's Remote Method Invocation (RMI). When a client and server wish to communicate over RMI, but the server wants to use SSL or some other protocol other than "no encryption", the client may not support the protocol the server would like to use. When that's the case, the client will generally download the proper socket implementation from the server at runtime. This constitutes a big security hole, because the server has yet to be authenticated at the time that the encryption interface is downloaded. That means an attacker can pretend to be the server, installing a malicious socket implementation on each client, even when the client already had proper SSL classes installed. The problem is that if the client fails to establish a secure connection with the default libraries (a failure), it will establish a connection using whatever protocol an untrusted entity gives it, thereby extending trust when it should not be extended.
>
> If your software has to fail, make sure it does so securely!

We include two relevant principles from Howard and LeBlanc [Howard 02] in Chapter 3, "Security Principles to Live By," in "Plan on Failure" on page 64:[10]

> As I've mentioned, stuff fails and stuff breaks. In the case of mechanical equipment, the cause might be wear and tear, and in the case of software and hardware, it might be bugs in the system. Bugs happen—plan on them occurring. Make security contingency plans. What happens if the firewall is breached? What happens if the Web site is defaced? What happens if the application is compromised? The wrong answer is, "It'll never happen!" It's like having an escape plan in case of fire—you hope to never have to put the strategy into practice, but if you do you have a better chance of getting out alive.
>
> Tip: Death, taxes, and computer system failure are all inevitable to some degree. Plan for the event.

And this from Chapter 3, "Security Principles to Live By," in "Fail to a Secure Mode" on pages 64-66:[11]

> So, what happens when you do fail? You can fail securely or insecurely. Failing to a secure mode means the application has not disclosed any data that would not be disclosed ordinarily, that the data still cannot be tampered with, and so on. Or you can fail insecurely such that the application discloses more than it should or its data can be tampered with (or worse). The former is the only proposition worth considering—if an attacker knows that he can make your code fail, he can bypass the security mechanisms because your failure mode is insecure.
>
> Also, when you fail, do not issue huge swaths of information explaining why the error occurred. Give the user a little bit of information, enough so that the user knows the request failed, and log the details to some secure log file, such as the Windows event log.
>
> Important: The golden rule when failing securely is to deny by default and allow only once you have verified the conditions to allow.

Example

10. From *Writing Secure Code, Second Edition* (0-7356-1722-8) by Microsoft Press. All rights reserved.

11. From *Writing Secure Code, Second Edition* (0-7356-1722-8) by Microsoft Press. All rights reserved.

For a microview of insecure failing, look at the following (pseudo) code and see whether you can work out the security flaw:

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == ERROR_ACCESS_DENIED) {
  // Security check failed.
  // Inform user that access is denied.
} else {
  // Security check OK.
}
```

At first glance, this code looks fine, but what happens if IsAccessAllowed fails? For example, what happens if the system runs out of memory, or object handles, when this function is called? The user can execute the privileged task because the function might return an error such as ERROR NOT ENOUGH MEMORY.

The correct way to write this code is as follows:

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == NO_ERROR) {
  // Secure check OK.
  // Perform task.
} else {
  // Security check failed.
  // Inform user that access is denied.
}
```

In this case, if the call to IsAccessAllowed fails for any reason, the user is denied access to the privileged operation.

A list of access rules on a firewall is another example. If a packet does not match a given set of rules, the packet should not be allowed to traverse the firewall; instead, it should be discarded. Otherwise, you can be sure there's a corner case you haven't considered that would allow a malicious packet, or a series of such packets, to pass through the firewall. The administrator should configure firewalls to allow only the packet types deemed acceptable though, and everything else should be rejected.

Another scenario, covered in detail in Chapter 10 [of *Writing Secure Code*], "All Input is Evil!" is to filter user input looking for potentially malicious input and rejecting the input if it appears to contain malevolent characters. A potential security vulnerability exists if an attacker can create input that your filter does not catch. Therefore, you should determine what is valid 'input and reject all other input."

According to Bishop [Bishop 03] in Chapter 13, "Design Principles," in Section 13.2.2, "Principle of Fail-Safe Defaults," on page 344:[12]

This principle restricts how privileges are initialized when a subject or object is created.

Definition 13-2. The Principle of Fail-Safe Defaults states that, unless a subject is given explicit access to an object, it should be denied access to that object.

This principle requires that the default access to an object is none. Whenever access, privileges, or some security-related attribute is not explicitly granted, it should be denied. Further, if the subject is unable to complete its action or task, before the subject terminates, it should undo those changes it made to the security state of the system. This way, even if the program fails, the system is still safe.

12. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

Example 1

If the mail server is unable to create a file in the spool directory, it should close the network connection, issue an error message, and stop. It should not try to store the message elsewhere, nor expand its privileges to save the message in another location because an attacker could use that ability to overwrite other files or fill up other disks (a denial of service attack). The protections on the mail spool directory itself should allow create and write access to only the mail server, and read and delete access to only the local server. No other user should have access to the directory.

In practice, most systems will allow an administrator access to the mail spool directory. By the principle of least privilege, that administrator should only be able to access the subjects and objects involved in mail queueing and delivery. As we saw, this minimizes the threats if that administrator's account is compromised. The mail system can be damaged or destroyed, but nothing else can be.

According to NIST [NIST 01] in Section 3.3, "IT Security Principles," on page 10:

Design and operate an IT system to limit vulnerability and to be resilient in response.

Information systems should be resistant to attack, should limit damage, and should recover rapidly when attacks do occur. The principle suggested here recognizes the need for adequate protection technologies at all levels to ensure that any potential cyber attack will be countered effectively. There are vulnerabilities that cannot be fixed, those that have not yet been fixed, those that are not known, and those that could be fixed but are not (e.g., risky services allowed through firewalls) to allow increased operational capabilities. In addition to achieving a secure initial state, secure systems should have a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state. Organizations should establish detect and respond capabilities, manage single points of failure in their systems, and implement a reporting strategy.

According to Schneier [Schneier 00] in "Security Processes":

Fail Securely.

Design your networks so that when products fail, they fail in a secure manner. When an ATM fails, it shuts down; it doesn't spew money out its slot.

"What Goes Wrong"

According to McGraw and Viega [McGraw 03]:[15]

A flaw in this DNS-spoofing detector dulled its paranoia.

If your software doesn't fail safely, you're in trouble. An example reported in August 2001 was found in versions 3.2 to 4.3 of the FreeBSD operating system, where the tcp_wrappers PARANOID hostname checking does not work properly. A flawed check for a numeric result during reverse DNS lookup caused the broken code in tcp_wrappers to skip some of its sanity checking for DNS results, allowing access to an attacker impersonating a trusted host (see the full explanation on the Bugtraq list at http://online.securityfocus.com/advisories/3515).

15. All rights reserved. It is reprinted with permission from CMP Media LLC.

References

[Bishop 03]	Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.
[Howard 02]	Howard, Michael & LeBlanc, David. *Writing Secure Code*. 2nd. Redmond, WA: Microsoft Press, 2002.
[McGraw 03]	McGraw, Gary & Viega, John. "Keep It Simple." *Software Development.* CMP Media LLC, May, 2003.
[NIST 01]	*Engineering Principles for Information Technology Security*. Special Publication 800-27. US Department of Commerce, National Institute of Standards and Technology, 2001.
[Saltzer 75]	Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63*, 9. IEEE, September 1975.
[Schneier 00]	Schneier, Bruce. "The Process of Security." *Information Security Magazine.* April, 2000. http://infosecuritymag.techtarget.com/archives2000.shtml#apr2000.
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

Pola

Nazwa	Warto##
Copyright Holder	Cigital, Inc.

Pola

1. mailto:copyright@cigital.com

Nazwa	Warto##
is-content-area-overview	false
Content Areas	Knowledge/Principles
SDLC Relevance	Design
Workflow State	Publishable